FUJITSU

FUJITSU Software NetCOBOL V11.0

Getting Started with COM Components

Windows

B1WD-3358-01ENZ0(00)
August 2015

Preface

Fujitsu NetCOBOL provides COM features based on Microsoft Technology. NetCOBOL supports COM features in COBOL object-oriented specifications. This manual describes how to create and use COBOL COM components.

Audience

The audience for this documentation is COBOL programmers responsible for building and delivering applications to users who do not have Fujitsu NetCOBOL installed on their machines.

How This Manual is Organized

This manual contains the following information:

Chapter 1	Introduction - describes COBOL object-oriented features.
Chapter 2	Converting a Legacy Procedural COBOL Program into an Object-Oriented COBOL Program.
Chapter 3	Creating and Using COBOL Components.
Chapter 4	Using COBOL COM Server with Visual Basic
Chapter 5	Using COBOL COM Server with ASP Pages

Note:

This manual contains product descriptions of some products that have been deprecated. Please refer to the product manual for additional information on product information and usage.

Conventions Used in This Manual

Example of convention	Description
setup	Characters you enter appear in bold.
Program-name	Underlined text indicates a placeholder for information you supply.
ENTER	Small capital letters are used for the name of keys and key sequences such as ENTER and CTRL+R. A plus sign (+) indicates a combination of keys.
…	Ellipses indicate the item immediately preceding can be specified repeatedly.
Edit, Literal	Names of menus and options appear with the initial letter capitalized.
[def]	Indicates that the enclosed item may be omitted.
{ABC|DEF}	Indicates that one of the enclosed items delimited by | is to be selected.
CHECK PARAGRAPH-ID COBOL ALL	Commands, statements, clauses, and options you enter or select appear in uppercase. Program section names, and some proper names also appear in uppercase. Underlined text indicates the default.
DATA DIVISION. WORKING-STORAGE SECTION. * Get the #INCLUDE file to the data	This font is used for examples of program code.

control #INCLUDE "NUMDATA.COB".	
The *form* acts as an application creation window.	Italics are occasionally used for emphasis.
"PowerCOBOL User's Guide" See Chapter 6, "Creating an Executable Program."	References to other publications or chapters within publications are in quotation marks.

Related Manuals

- NetCOBOL Getting Started
- NetCOBOL User's Guide
- NetCOBOL Language Reference
- NetCOBOL Debugging Guide
- PowerCOBOL User's Guide
- PowerCOBOL Reference
- PowerFORM Getting Started

Trademarks

- Microsoft, Active X, Visual Basic, Windows, Windows Server, and Visual Studio are either trademarks or registered trademarks of Microsoft Corporation in the U.S.A. and/or other countries.
- NetCOBOL is a trademark or registered trademark of Fujitsu Limited or its subsidiaries in the United States or other countries or in both.
- Adobe, Acrobat, Acrobat Reader, and Acrobat logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.
- Other product names are trademarks or registered trademarks of each company.
- Trademark indications are omitted for some system and product names described in this manual.

Security

Since there are Internet-enabled functions in the development environment, it is recommended that you use the NetCOBOL development environment only in an Intranet, and not in the more open Internet environment.

To ensure security when working in an environment that is connected to the Internet, it is important to correctly set up both the applications created with NetCOBOL and their operating environment.

To safeguard resources (such as databases, and input and output files), and definition and information files required for the operation of programs from illegal access and tampering, you need to restrict access to the resources by OS functions and programs. In particular, keep important resources in an intranet environment in which a firewall has been installed.

Although this product offers different communication functions (such as the simple communication interface facility, and the Web subroutines), only the Web subroutines have been designed for use with Internet services. Therefore, only use these other functions in environments that are not connected to the Internet, or in intranet environments in which firewalls have been installed and which have been constructed to prevent security breaches.

If you are using the Web subroutines with NetCOBOL on a Web server, use the Web server authorization apparatus and encryption communication function (SSL) to prevent illegal access or information being leaked or tampered with. Additionally, use the Web server access log to investigate and pursue any incidents of illegal access. For details, refer to the documentation for the Web server you are using.

You need to test applications created with NetCOBOL to ensure that even if malicious or careless data values are provided as input, no important data can be destroyed or sensitive information obtained.

High Risk Activity

The Customer acknowledges and agrees that the Product is designed, developed and manufactured as contemplated for general use, including without limitation, general office use, personal use, household use, and ordinary industrial use, but is not designed, developed and manufactured as contemplated for use accompanying fatal risks or dangers that, unless extremely high safety is secured, could lead directly to death, personal injury, severe physical damage or other loss (hereinafter "High Safety Required Use"), including without limitation, nuclear reaction control in nuclear facility, aircraft flight control, air traffic control, mass transport control, medical life support system, missile launch control in weapon system.

The Customer shall not use the Product without securing the sufficient safety required for the High Safety Required Use. In addition, Fujitsu (or other affiliate's name) shall not be liable against the Customer and/or any third party for any claims or damages arising in connection with the High Safety Required Use of the Product.

Export Regulation

Exportation/release of this software may require necessary procedures in accordance with the regulations of your resident country and/or US export control laws.

Product names

The names of products described in this manual are abbreviated as follows:

Product Name	Abbreviation
Microsoft® Windows Server® 2012 R2 Datacenter Microsoft® Windows Server® 2012 R2 Standard Microsoft® Windows Server® 2012 R2 Essentials Microsoft® Windows Server® 2012 R2 Foundation	Windows Server 2012 R2
Microsoft® Windows Server® 2012 Datacenter Microsoft® Windows Server® 2012 Standard Microsoft® Windows Server® 2012 Essentials Microsoft® Windows Server® 2012 Foundation	Windows Server 2012
Microsoft® Windows Server® 2008 R2 Foundation Microsoft® Windows Server® 2008 R2 Standard Microsoft® Windows Server® 2008 R2 Enterprise Microsoft® Windows Server® 2008 R2 Datacenter	Windows Server 2008 R2
Windows® 8.1 Windows® 8.1 Pro Windows® 8.1 Enterprise	Windows 8.1
Windows® 8 Windows® 8 Pro Windows® 8 Enterprise	Windows 8
Windows® 7 Home Premium Windows® 7 Professional Windows® 7 Enterprise Windows® 7 Ultimate	Windows 7

Microsoft Windows products listed in the table above are referred to in this manual as "Windows".

August 2015

Contents

Chapter 1 Introduction

Microsoft's COM (Component Object Model) is a specification by which development environments can create and use common modules in the Microsoft Windows environment. These modules share a standard set of data types and a common programming interface. This allows modules written in Microsoft Visual Basic or Visual C++ (for example) to utilize modules written in NetCOBOL or vice-versa.

Examples of COM technologies include Microsoft's OLE (object Linking and Embedding) and the later replacement for OLE known as ActiveX

COM-enabled development environments generally require that you program using an object-oriented methodology. We will discuss this a bit later in this manual.

Fujitsu's PowerCOBOL graphical user interface (GUI) development environment allows users to create single-threaded graphical COM components (typically referred to as "controls"). It does not currently support the creation of multi-threaded COM controls, however.

One of the advantages of using PowerCOBOL to create COM components (controls) is the fact that you are not required to learn object-oriented COBOL programming skills and design. PowerCOBOL allows you to create these modules by programming in traditional procedural COBOL, with a few exceptions.

You may use PowerCOBOL to create any graphical COBOL COM controls that you might require. You should refer to the Fujitsu PowerCOBOL documentation for information on how to create these types of controls. Note that when you use PowerCOBOL to create graphical COM controls, there is additional overhead required to support these in the form of the PowerCOBOL runtime system.

This manual is dedicated to creating non-graphical COBOL COM components using the NetCOBOL development environment. It will not touch on PowerCOBOL any further.

COM components created using Fujitsu's NetCOBOL development environment require that the base client runtime support files be installed on any client machine you may wish to distribute them to. The NetCOBOL development environment already includes these files so there is no need to install the client runtime system on a machine containing such. In fact, you should not attempt to install the NetCOBOL client runtime on a machine already containing the NetCOBOL development environment.

COBOL COM components can be built to use Fujitsu's royalty free single-threaded COBOL runtime system or Fujitsu's non-royalty free multi-threaded runtime system.

This manual will illustrate how to create both single and multi- threaded COBOL COM components. In most cases, single- threaded COBOL COM components will work fine in your applications. In other cases, such as running on high transaction processing web servers, you may wish to purchase a multi-threaded runtime license from Fujitsu for higher performance and better compatibility. Contact your Fujitsu sales representative or visit NetCOBOL's web site at http://www.adtools.com for details.

The following discussion is intended as a brief introduction to object-oriented terminology. If you are already familiar with OO terminology, you may wish to skip this discussion.

1.1 Object-Oriented COBOL Usage and Terminology

NetCOBOL supports the current ANSI COBOL Object -oriented specification. This means that you can write object -oriented COBOL programs using NetCOBOL.

Creating COM components using the NetCOBOL development environment requires that you create OO COBOL programs.

This manual will not attempt to teach you object-oriented COBOL (OO COBOL). In the next chapter, however, it will detail how to convert a procedural COBOL program into an OO COBOL program. For in-depth information about programming in OO COBOL, refer to the NetCOBOL documentation. You may additionally examine the NetCOBOL sample programs that are by default installed in:

C:\Program Files\Fujitsu NetCOBOL for Windows\COBOL\Samples

The OO COBOL programming examples begin with SAMPLE15 and go up.

Some of the object-oriented terms you need to understand are:

- Class - A Class is a definition of data and functions to act upon the data. In COBOL terms, it is the high level definition of an executable module or OO COBOL program.
- Object - An Object is a specific executable program instance derived from a Class. In COBOL terms, it is a specific executable load module (Window's .DLL) created from a COBOL class at runtime.

- Inheritance - Inheritance is the concept of building a class by using properties and methods from an existing class as a base and then adding to it. The new class inherits the properties and methods of an existing class.
- Method - A Method is a specific function contained within an Object. In COBOL terms, it is a callable function (for example, a Performable COBOL Paragraph) contained within an OO COBOL Program.
- Property - A Property is data item that contains a given value. In COBOL terms, it is a data item that may or may not be exposed outside the COBOL program.

1.2 Classes and Objects

In OO COBOL, a Class is a COBOL program written with a class identifier. We will look at examples of classes in the next chapter. At runtime, a program that desires to call or connect to a COBOL Class will create a new instance of this class first. This instance of the COBOL Class is known as an Object.

In OO COBOL programming, an OO COBOL Class can have an object instantiated by a traditional procedural COBOL program using just a bit of OO COBOL syntax.

When an OO COBOL class is called to instantiate an object (remember that the object is the runtime executable instance of the OO COBOL program defined in the OO COBOL class), it returns a pointer (sometimes called a handle) to the newly created object. This pointer is stored in a COBOL data item defined as USAGE OBJECT REFERENCE.

Figure 1.1 shows a traditional procedural COBOL program that instantiates a COBOL class named FILEIO4 and executes various Methods in it.

```
Identification Division.
Program-ID. TESTFILEIO4.
Environment Division.
Configuration Section.
Repository.
    Class FILEIO4.
Data Division.
Working-Storage Section.
01 FILEIO-PTR Usage Object Reference.
01 File-Operation Pic X(5).
01 Return-Status Pic 99 Value 0.
01 Return-Record Pic X(132).
Procedure Division.
    Invoke FILEIO4 "NEW" Returning FILEIO-PTR.

    Invoke FILEIO-PTR "OPEN-FILE" Using File-Operation
                                        Return-Status
                                        Return-Record.
    If Return-Status = 0
        Perform Until Return-Status Not = 0
            Invoke FILEIO-PTR "READ-FILE" Using File-Operation
                                                Return-Status
                                                Return-Record
            If Return-Status = 0
                Display Return-Record
            End-If
        End-Perform
        Invoke FILEIO-PTR "CLOSE-FILE" Using File-Operation
                                             Return-Status
                                             Return-Record
    End-If.
    Set FILEIO-PTR To Null.
    Exit Program.
```

Figure 1.1 A sample COBOL program instantiating an OO COBOL class

In examining the program in Figure 1.1, you will note the insertion of a Repository under the ENVIRONMENT DIVISION's CONFIGURATION SECTION. The Repository specifies the name of any OO COBOL classes that the program may reference at runtime.

In the PROCEDURE DIVISION, you will notice the following statement:

```
Invoke FILEIO4 "NEW" Returning FILEIO-PTR.
```

Invoke is an OO COBOL verb that is similar to a CALL statement. Invoke will always call an OO COBOL Class (or other language COM or OLE object) and execute a specific Method. The Method being Invoked (called) in the previous example is the "NEW" method, which creates a new instance (executable) of the FILEIO4 class (program). This statement returns a pointer to the new instance of the class, which is stored in the WORKING-STORAGE data item:

```
01 FILEIO-PTR Usage Object Reference.
```

If you examine the procedural code further, you will note additional invoke statements such as:

```
Invoke FILEIO-PTR "READ-FILE" Using File-Operation
                                    Return-Status
                                    Return-Record.
```

Invoking FILEIO-PTR actually calls the instance of the FILEIO4 class that was created with the invocation of its "NEW" method. The previous example is now calling the "READ-FILE" method of the FILEIO4 object that was created and is passing three parameters to it via the USING statement, which is identical to the way parameters are passed in a traditional CALL statement.

Just before the program is exited, you will note the following statement:

```
Set FILEIO-PTR To Null.
```

By setting the OBJECT REFERENCE named FILEIO-PTR to Null, we release this instance of the class (release the object), thus deleting it from memory. It is important to do this or you can leave orphan objects laying around in memory. In COBOL terms, this is similar to executing a CANCEL statement.

Let's have a look at the actual OO COBOL class (program) being instantiated in this example (FILEIO4).

Figure 1.2 contains a listing of the FILEIO4 OO COBOL program.

```
Identification Division.
Class-ID. FILEIO4 Inherits FJBASE.
Environment Division.
Configuration Section.
Repository.
    Class FJBASE.
Object.
Environment Division.
Input-Output Section.
File-Control.
    Select Infile Assign To "test.txt"
                  Organization is Line Sequential
                  File Status is Infile-Status.
Data Division.
File Section.
FD Infile.
01 Infile-Record Pic X(132).
Working-Storage Section.
01 Infile-Status Pic 99.
01 File-Opened-Flag Pic 9 Value 0.
Procedure Division.

Method-ID. OPEN-FILE.
Data Division.
Linkage Section.
01 File-Operation Pic X(5).
01 Return-Status Pic 99.
01 Return-Record Pic X(132).
Procedure Division Using File-Operation
```

```
                        Return-Status
                        Return-Record.
    If File-Opened-Flag Not = 0
        *> File is already open, so return error
        Move 99 To Return-Status
    Else
        Open Input Infile
        Move Infile-Status To Return-Status
        If Infile-Status = 0
            Move 1 To File-Opened-Flag
        End-If
    End-If.
End Method OPEN-FILE.

Method-ID. CLOSE-FILE.
Data Division.
Linkage Section.
01 File-Operation Pic X(5).
01 Return-Status Pic 99.
01 Return-Record Pic X(132).
Procedure Division Using File-Operation.
                        Return-Status
                        Return-Record.
    If File-Opened-Flag = 0
        *> File is not open, so return error
        Move 88 To Return-Status
    Else
        Close Infile
        Move Infile-Status To Return-Status
        Move 0 To File-Opened-Flag
    End-If.
End Method CLOSE-FILE.

Method-ID. READ-FILE.
Data Division.
Linkage Section.
01 File-Operation Pic X(5).
01 Return-Status Pic 99.
01 Return-Record Pic X(132).
Procedure Division using File-Operation
                        Return-Status
                        Return-Record.
    If File-Opened-Flag = 0
        *> File is not open, so return error
        Move 88 To Return-Status
    Else
        Read Infile
        Move Infile-Status to Return-Status
        Move Infile-Record to Return-Record
    End-If.
End Method READ-FILE.

End Object.
End Class FILEIO4.
```

Figure 1.2 The FILEIO4 OO COBOL Class (Program)

1.3 Inheritance

By examining the FILEIO4 class source code, you'll first notice that instead of a PROGRAM-ID statement, we have a CLASS-ID statement as follows:

```
Class-ID. FILEIO4 Inherits FJBASE.
```

The Inherits statement is another object-oriented term. In this case, it means that the class named FILEIO4 inherits all of the methods and properties from the FJBASE class. Inheritance is a very powerful feature of object-oriented programming. It means that a class can be created on top of another class, adding all of the inherited class' functionality (properties and methods) into it automatically.

All Fujitsu OO COBOL classes typically inherit from the FJBASE base class, which is the Base class for Fujitsu OO COBOL. This saves a lot of time and trouble in developing applications.

For example, the program shown in Figure 1.1 is calling a method in the FILEIO4 class named "NEW" to create a new instance of it at runtime. If you examine the actual source code for the FILEIO4 class, however, you will note that there is no method definition named "NEW" in this class. The "NEW" method is actually contained in the FJBASE class and because the FILEIO4 class inherits from the FJBASE class, the "NEW" method is available for the FILEIO4 class.

1.4 Methods

By examining the OO COBOL program named FILEIO4 in Figure 1.2, you will note three separate Method definitions - OPEN-FILE, READ-FILE and CLOSE-FILE. These methods look almost like mini COBOL programs within the main COBOL class, and you may think of them as such. They have their own DATA DIVISIONS and LINKAGE SECTIONS and they could also have their own WORKING-STORAGE SECTIONS if needed.

Any data items defined within a method are local only to that method and cannot be referenced by other methods in the same class or by any other class or program invoking the method.

1.5 Properties

Properties are data items that are defined in the class' WORKING-STORAGE SECTION. Properties may be either Public or Private. In the sample OO COBOL program shown in Figure 1.2, the following definitions are property definitions.

```
01 Infile-Status Pic 99.
01 File-Opened-Flag Pic 9 Value 0.
```

These properties may be referenced by any method within the class and may thus be thought of as global data items within the class. By default, WORKING-STORAGE SECTION data items defined at the class level are Private properties. Private properties are not exposed to the outside world and thus cannot be referenced from outside programs. To create a Public property that would be exposed to outside programs, we would simply add the "Property" key word in the definition such as.

```
01 Infile-Status Pic 99 Property.
01 File-Opened-Flag Pic 9 Value 0 Property.
```

Chapter 2 Converting a Legacy Procedural COBOL Program into an Object-Oriented COBOL Program

In this chapter, we will discuss how to convert a traditional legacy procedural COBOL program into an Object-Oriented (OO) COBOL program.

If you are already familiar with OO COBOL programming, you may wish to skip this chapter.

When you use the NetCOBOL development environment to create COM modules from COBOL programs, it is required that these programs be OO COBOL programs. This means that if you wish to migrate traditional procedural COBOL programs to COM, you must first restructure them as OO COBOL programs.

In some cases, this can be as simple as adding a few lines of code at the beginning and end of a COBOL program, defining it as a Class, and adding some code around the main portion of the program to create a method definition.

In other cases, you may need to or desire to break the program up into multiple method definitions for functionality reasons. In the examples in this chapter, we are going to look at what is required to take a simple file I/O module written in procedural COBOL and change it into an OO COBOL program.

The program we are going to start working with is shown in Figure 2.1.

```
Program-ID. FILEIO.
Environment Division.
Input-Output Section.
File-Control.
    Select Infile Assign To "test.txt"
                  Organization is Line Sequential
                  File Status is Infile-Status.
Data Division.
File Section.
FD Infile.
01 Infile-Record Pic X(132).
Working-Storage Section.
01 Infile-Status Pic 99.
01 File-Opened-Flag Pic 9 Value 0.
Linkage Section.
01 File-Operation Pic X(5).
01 Return-Status Pic 99.
01 Return-Record Pic X(132).
Procedure Division Using File-Operation
                         Return-Status
                         Return-Record.
    Evaluate File-Operation
        When "OPEN" Perform Open-File
        When "CLOSE" Perform Close-File
        When "READ" Perform Read-File
    End-Evaluate.
    Exit Program.

Open-File.
    If File-Opened-Flag Not = 0
        *> File is already open, so return error
        Move 99 To Return-Status
    Else
        Open Input Infile
        Move Infile-Status To Return-Status
        If Infile-Status = 0
            Move 1 To File-Opened-Flag
        End-If
    End-If
Open-File-Exit.
```

```
    Exit.

Close-File.
    If File-Opened-Flag = 0
        *> File is not open, so return error
        Move 88 To Return-Status
    Else
        Close Infile
        Move Infile-Status To Return-Status
        Move 0 To File-Opened-Flag
    End-If.
Close-File-Exit.
    Exit.

Read-File.
    If File-Opened-Flag = 0
        *> File is not open, so return error
        Move 88 To Return-Status
    Else
        Read Infile
        Move Infile-Status To Return-Status
        Move Infile-Record to Return-Record
    End-If.
Read-File-Exit.
    Exit.
```

Figure 2.1 A traditional procedural COBOL program

The program named FILEIO shown in Figure 2.1 is a simple callable program that manages the opening, reading, and closing of a text file. It expects to be called with three LINKAGE-SECTION parameters:

- FILE-OPERATION - Specifies which file I/O operation is being requested. Valid values are "OPEN", "READ", and "CLOSE".
- RETURN-STATUS - Returns the file status as a result of attempting the file I/O operation being requested.
- RETURN-RECORD - Returns the record that was read when a successful "READ" file I/O operation takes place.

This program has a main EVALUATE loop that checks the file I/O operation being requested in the FILE-OPERATION parameter being passed in and performs one of three separate paragraphs that correspond to the “OPEN”, “READ” and “CLOSE” file I/O operations.

A WORKING-STORAGE SECTION data item named "File-Opened-Flag" maintains the current Open/Close state of the text file and is used for error checking purposes. For example, a “READ” request that comes before the file is successfully opened should raise an error.

The FILEIO program will be called by another program to first OPEN the file, then to READ all of the records and then to CLOSE the file. When the last record is read, a file status will be raised by the runtime system and the calling program will thus be able to check for this to know when the end of the file has been reached.

Figure 2.2 shows a main program named “TESTFILEIO” that will call the FILEIO subprogram.

```
Identification Division.
Program-ID. TESTFILEIO.
Environment Division.
Data Division.
Working-Storage Section.
01 File-Operation Pic X(5).
01 Return-Status Pic 99 Value 0.
01 Return-Record Pic X(132).
Procedure Division.
    Move "OPEN" To File-Operation.
    Call "FILEIO" Using File-Operation
                        Return-Status
                        Return-Record.
    If Return-Status = 0
        Move "READ" To File-Operation
```

```
        Perform Until Return-Status Not = 0
            Call "FILEIO" Using File-Operation
                                Return-Status
                                Return-Record
            If Return-Status = 0
                Display Return-Record
            End-If
        End-Perform
        Move "CLOSE" To File-Operation
        Call "FILEIO" Using File-Operation
                            Return-Status
                            Return-Record
    End-If.
    Exit Program.
```

Figure 2.2 The TESTFILEIO main program that will call the FILEIO subprogram

The TESTFILEIO program is also quite simple for the purposes of this illustration. It calls the FILEIO subprogram to first OPEN the text file. It then performs a loop calling FILEIO again to read each record and displays the returned records. Finally upon reaching the end of the file (determined by checking the file status code returned in the RETURN-STATUS parameter), it requests FILEIO to close the file.

Figure 2.3 shows the output of having executed the TESTFILEIO program that called the FILEIO subprogram reading a simple text file containing four records.

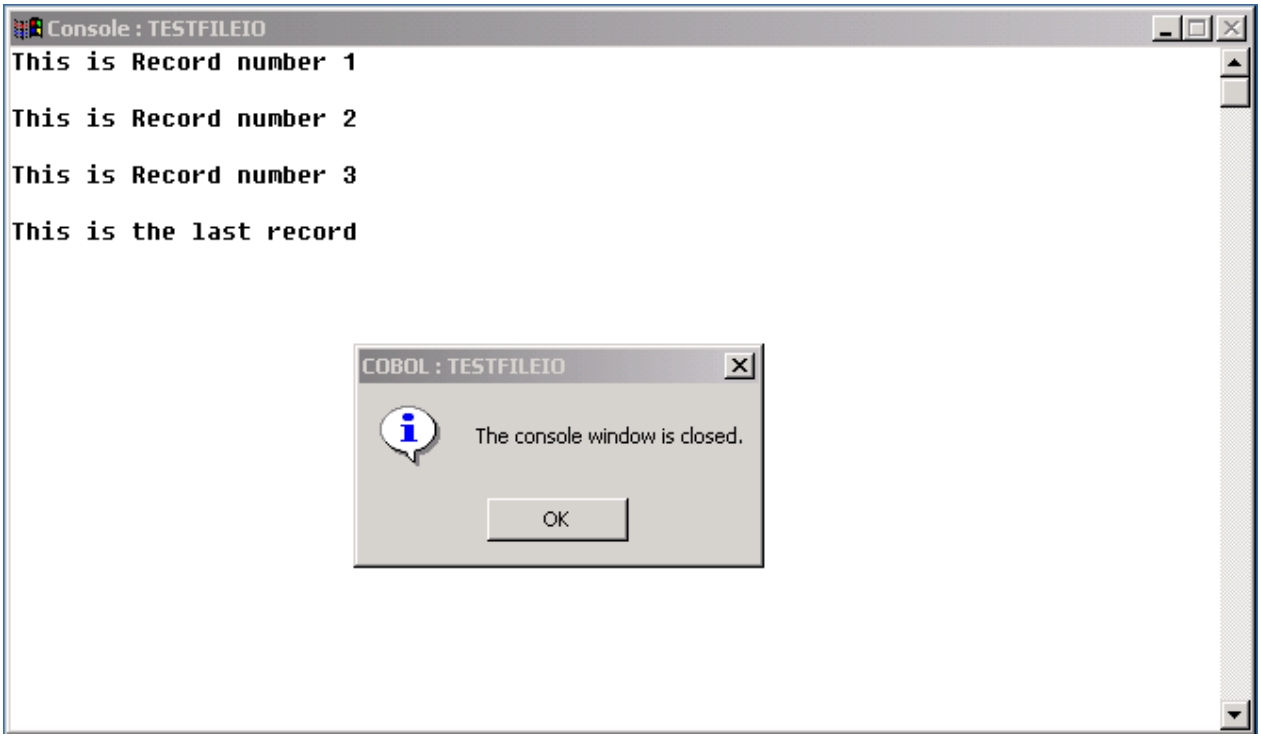

Figure 2.3 The output of executing the TESTFILEIO program

You may examine the FILEIO and TESTFILEIO programs in detail and execute them by loading the FILEIO.PRJ project.

We will now examine the various approaches to converting the FILEIO program into an Object-Oriented COBOL program. We will also make changes to the main TESTFILEIO program to allow it to utilize the newly converted FILEIO program

2.1 Converting the FILEIO Program into a Simple Object-Oriented COBOL Program

Converting the FILEIO program shown in Figure 2.1 requires some knowledge of OO COBOL, but is also a reasonably straightforward process.

The project file named FILEIO2.PRJ contains the programs we will refer to in the following text. You may use it to examine and execute the application as converted in this section.

The main TESTFILEIO program will also require a couple of modifications in order to call the OO COBOL version of the FILEIO program that will be named FILEIO2. In this first example, however, TESTFILEIO will not be converted into an OO COBOL program itself. Instead, it will be OO enabled to call the FILEIO2 OO COBOL program.

This implies that you can create applications that use both traditional procedural COBOL programs and OO COBOL programs together. A traditional procedural COBOL program requires a few minor modifications in order to call an OO COBOL program and must use the INVOKE statement instead of the traditional CALL statement in order to do so.

An OO COBOL program can call a traditional procedural COBOL program using the standard CALL statement. A traditional procedural COBOL program does not require any modification in order to be called by an OO COBOL program. Modifications to a traditional procedural COBOL program are only required when such a program wishes to call an OO COBOL program.

Figure 2.4 shows the results of converting the FILEIO COBOL program into a simple object-oriented COBOL program named FILEIO2.

```
Identification Division.
Class-ID. FILEIO2 Inherits FJBASE.
Environment Division.
Configuration Section.
Repository.
    Class FJBASE.

Object.
Environment Division.
Input-Output Section.
File-Control.
    Select Infile Assign To "test.txt"
                  Organization is Line Sequential
                  File Status is Infile-Status.
Data Division.
File Section.
FD Infile.
01 Infile-Record Pic X(132).
Working-Storage Section.
01 Infile-Status Pic 99.
01 File-Opened-Flag Pic 9 Value 0.
Procedure Division.

Method-ID. CALL_FILEIO2.
Data Division.
Linkage Section.
01 File-Operation Pic X(5).
01 Return-Status Pic 99.
01 Return-Record Pic X(132).
Procedure Division Using File-Operation
                         Return-Status
                         Return-Record.
    Evaluate File-Operation
        When "OPEN" Perform Open-File
        When "CLOSE" Perform Close-File
        When "READ" Perform Read-File
    End-Evaluate.
    Exit Method.

Open-File.
```

```
    If File-Opened-Flag Not = 0
        *> File is already open, so return error
        Move 99 To Return-Status
    Else
        Open Input Infile
        Move Infile-Status To Return-Status
        If Infile-Status = 0
            Move 1 To File-Opened-Flag
        End-If
    End-If.
Open-File-Exit.
    Exit.

Close-File.
    If File-Opened-Flag = 0
        *> File is not open, so return error
        Move 88 To Return-Status
    Else
        Close Infile
        Move Infile-Status To Return-Status
        Move 0 To File-Opened-Flag
    End-If.
Close-File-Exit.
    Exit.

Read-File.
    If File-Opened-Flag = 0
        *> File is not open, so return error
        Move 88 To Return-Status
    Else
        Read Infile
        Move Infile-Status to Return-Status
        Move Infile-Record to Return-Record
    End-If.
Read-File-Exit.
    Exit.
End Method CALL_FILEIO2.

End Object.
End Class FILEIO2.
```

Figure 2.4 The FILEIO program converted to an OO COBOL program and renamed FILEIO2

If you compare the original FILEIO program shown in Figure 2.1 to the OO COBOL version of it named FILEIO2 shown in Figure 2.4, you will notice the following changes:

1. The Program-ID statement has been changed to a Class-ID statement and reflects the name change (FILEIO2). It additionally contains the phrase: Inherits FJBASE (Refer to the topic entitled "Inheritance" in Chapter One for an explanation).
2. A CONFIGURATION SECTION has been added to the ENVIRONMENT DIVISION.
3. A Repository statement has been added to the newly inserted CONFIGURATION SECTION. The Repository is use to list any outside classes this class may refer to.
4. A Class reference to FJBASE has been added to the newly inserted Repository.
5. An Object statement has been added after the Repository. This defines the start of the actual object or program that will be instantiated when the FJBASE Class' "NEW" method is invoked.
6. An ENVIRONMENT DIVISION has been added for the Object. Note that the original ENVIRONMENT DIVISION from FILEIO has been left intact after this point. Also notice that the DATA DIVISION has been left intact as well, following this point, with the exception of the LINKAGE SECTION, which has been moved down into a method.

7. We've added a PROCEDURE DIVISION statement for the object itself, and we have inserted after it a Method-ID statement and named the new method "CALL_FILEIO2". Every class must have a least one method, and you can think of it in this example as being similar to an Entry Point. A Class can have multiple methods and each must have a unique name and may be invoked (called). Methods look very much like individual nested COBOL programs.
8. We now define a new DATA DIVISION for this specific method, and the original LINKAGE SECTION from FILEIO has been placed here.
9. From this point forth, we have the identical PROCEDURE DIVISION USING … statement and all of the original FILEIO program's procedural logic, with the exception that "Exit Program" has been changed to Exit Method".
10. Finally, we've added three statements at the bottom of the program to close it off properly:
    a. End Method CALL_FILEIO2. - Ends the method definition.
    b. End Object. - Ends the object definition.
    c. End Class FILEIO2. - Ends the Class (Program) definition.

This is all that is needed to convert the traditional procedural COBOL program FILEIO to the OO COBOL program FILEIO2.

It's important to realize that the actual procedural code remained largely intact and did not require changes, with the lone exception of Exit Program to exit Method.

Let's look now at the changes required to the main TESTFILEIO program shown in Figure 2.2 to allow it to call our newly created FILEIO2 OO COBOL program.

Figure 2.5 shows the modified TESTFILEIO program that has been named TESTFILEIO2.

```
Identification Division.
Program-ID. TESTFILEIO2.
Environment Division.
Configuration Section.
Repository.
    Class FILEIO2.
Data Division.
Working-Storage Section.
01 FILEIO-PTR Object Reference.
01 File-Operation Pic X(5).
01 Return-Status Pic 99 Value 0.
01 Return-Record Pic X(132).
Procedure Division.
    Invoke FILEIO2 "NEW" Returning FILEIO-PTR.
    Move "OPEN" To File-Operation.
    *> Call "FILEIO" Using File-Operation
    Invoke FILEIO-PTR "CALL_FILEIO2" Using File-Operation
                                           Return-Status
                                           Return-Record.
    If Return-Status = 0
        Move "READ" To File-Operation
        Perform Until Return-Status Not = 0
            *> Call "FILEIO" Using File-Operation
            Invoke FILEIO-PTR "CALL_FILEIO2" Using File-Operation
                                                   Return-Status
                                                   Return-Record
            If Return-Status = 0
                Display Return-Record
            End-If
        End-Perform
        Move "CLOSE" To File-Operation
        *> Call "FILEIO" Using File-Operation
        Invoke FILEIO-PTR "CALL_FILEIO2" Using File-Operation
                                               Return-Status
                                               Return-Record
    End-If.
```

```
    Set FILEIO-PTR To Null.
    Exit Program.
```

Figure 2.5 The modified TESTFILEIO2 program

If you compare the original TESTFILEIO program shown in Figure 2.2 to the modified version named TESTFILEIO2 shown in Figure 2.5, you will note the following modifications:

1. A CONFIGURATION SECTION has been added in the ENVIRONMENT DIVISION.
2. A Repository has been added to the newly inserted CONFIGURATION SECTION.
3. A Class reference to the FILEIO2 class (program) we just modified has been added to the Repository.
4. In the WORKING-STORAGE SECTION, we've added a new data item which will hold the pointer to the FILEIO object we'll create a runtime. This data item is named:

   ```
   01 FILEIO-PTR Object Reference.
   ```

5. In the PROCEDURE DIVISION, the following Invoke statement has been added to create a new instance of the FILEIO2 class (program) and return the pointer to such in the FILEIO-PTR data item:

   ```
   Invoke FILEIO2 "NEW" Returning FILEIO-PTR.
   ```

6. The previous CALL statements to the FILEIO program have been commented out and replaced with:

   ```
   Invoke FILEIO-PTR "CALL_FILEIO2" Using File-Operation Return-Status Return-Record.
   ```

   Note that the syntax of the Invoke statement makes use of the identical USING statement and the same parameter list.

7. 7. Just before the EXIT PROGRAM statement, we've added a statement to release the class object (delete it) from memory:

   ```
   Set FILEIO-PTR To Null.
   ```

These are the only changes required to the original TESTFILEIO program to allow it to call the new FILEIO2 OO COBOL program. If you execute this application (contained in the FILEIO2.PRJ project), you will see the same results as having executed the original FILEIO.PRJ. These results are shown in Figure 2.3. We have now made the minimal required changes to the FILEIO program to turn it into an OO COBOL program.

2.2 Converting the FILEIO2 OO COBOL Program into a more Sophisticated Object-Oriented COBOL Program

We will now look at modifying the newly created FILEIO2 OO COBOL program into a more sophisticated OO COBOL program. We will use the FILEIO3.PRJ project as a reference for the following discussion.

When the original FILEIO program was modified to make it a simple OO COBOL program, the intent was to make a few changes as possible to it to accomplish the task of making it a valid OO COBOL program.

As we can see from executing the TESTFILEIO2 program, the behavior of the FILEIO2 OO COBOL program is the same as the behavior of the original FILEIO program.

In some cases when moving existing legacy applications into the OO COBOL world (remember that creating COM modules in COBOL 97 requires that they be OO COBOL programs), making the minimal set of changes will accomplish the task.

In the following example, however, we are going to make some changes to FILEIO2's actual structure that lends better to object- oriented methodology. Specifically, we are going to get rid of the Evaluate loop logic and break each of the three paragraphs out into Separate Methods - one to OPEN the file, another method to READ the file, and finally a third method to CLOSE the file.

We will then modify the TESTFILEIO2 program to call these three separate methods.

Figure 2.6 shows the FILEIO2 program modified as previously described. The new version of this program has been named FILEIO3.

```
Identification Division.
Class-ID. FILEIO3 Inherits FJBASE.
```

```
Environment Division.
Configuration Section.
Repository.
    Class FJBASE.

Object.
Environment Division.
Input-Output Section.
File-Control.
    Select Infile Assign To "test.txt"
                  Organization is Line Sequential
                  File Status is Infile-Status.
Data Division.
File Section.
FD Infile.
01 Infile-Record Pic X(132).
Working-Storage Section.
01 Infile-Status Pic 99.
01 File-Opened-Flag Pic 9 Value 0.
Procedure Division.

Method-ID. OPEN-FILE.
Data Division.
Linkage Section.
01 File-Operation Pic X(5).
01 Return-Status Pic 99.
01 Return-Record Pic X(132).
Procedure Division Using File-Operation
                         Return-Status
                         Return-Record.

Open-File.
    If File-Opened-Flag Not = 0
        *> File is already open, so return error
        Move 99 To Return-Status
    Else
        Open Input Infile
        Move Infile-Status To Return-Status
        If Infile-Status = 0
            Move 1 To File-Opened-Flag
        End-If
    End-If.
Open-File-Exit.
    Exit.
End Method OPEN-FILE.

Method-ID. CLOSE-FILE.
Data Division.
Linkage Section.
01 File-Operation    Pic X(5).
01 Return-Status    Pic 99.
01 Return-Record    Pic X(132).
Procedure Division Using File-Operation
                         Return-Status
                         Return-Record.

Close-File.
    If File-Opened-Flag = 0
        *> File is not open, so return error
        Move 88 To Return-Status
    Else
        Close Infile
        Move Infile-Status To Return-Status
```

```
        Move 0 To File-Opened-Flag
    End-If.
Close-File-Exit.
    Exit.
End Method CLOSE-FILE.

Method-ID. READ-FILE.
Data Division.
Linkage Section.
01 File-Operation    Pic X(5).
01 Return-Status    Pic 99.
01 Return-Record    Pic X(132).
Procedure Division Using File-Operation
                         Return-Status
                         Return-Record.

Read-File.
    If File-Opened-Flag = 0
        *> File is not open, so return error
        Move 88 To Return-Status
    Else
        Read Infile
        Move Infile-Status to Return-Status
        Move Infile-Record to Return-Record
    End-If.
Read-File-Exit.
    Exit.
End Method READ-FILE.

End Object.
End Class FILEIO3.
```

Figure 2.6 The FILEIO3 program

If you compare the FILEIO3 program shown in Figure 2.6 to the FILEIO2 program shown in Figure 2.4, you will notice the following modifications:

1. The Class-ID has been changed to FILEIO3.
2. The main Evaluate loop logic has been removed and we have deleted the CALL_FILEIO2 method and create three new methods named “OPEN-FILE”, “READ-FILE” and “CLOSE-FILE”.

Specifically, if you examine the three new methods, you’ll note they contain identical LINKAGE SECTIONS and PROCEDURE DIVISION USING… statements. The original procedural logic contained in these three new methods (which used to be 3 separate paragraphs) is identical. We have simply placed some code around each of the individual paragraphs to turn them into separate methods.

We now need to modify the TESTFILEIO2 program to call these new methods. The converted program is now named TESTFILEIO3 and is shown in Figure 2.7.

```
Identification Division.
Program-ID. TESTFILEIO3.
Environment Division.
Configuration Section.
Repository.
    Class FILEIO3.
Data Division.
Working-Storage Section.
01 FILEIO-PTR Object Reference.
01 File-Operation Pic X(5).
01 Return-Status Pic 99 Value 0.
01 Return-Record Pic X(132).
Procedure Division.
    Invoke FILEIO3 "NEW" Returning FILEIO-PTR.
```

```
    Invoke FILEIO-PTR "OPEN-FILE" Using File-Operation
                                        Return-Status
                                        Return-Record.
    If Return-Status = 0
        Perform Until Return-Status Not = 0
            Invoke FILEIO-PTR "READ-FILE" Using File-Operation
                                                Return-Status
                                                Return-Record
            If Return-Status = 0
                Display Return-Record
            End-If
        End-Perform
        Invoke FILEIO-PTR "CLOSE-FILE" Using File-Operation
                                             Return-Status
                                             Return-Record
    End-If.
    Set FILEIO-PTR To Null.
    Exit Program.
```

Figure 2.7 The TESTFILEIO3 program

Note that the only modifications to the TESTFILEIO3 program are:

1. We no longer need to specify the File-Operation parameter. We could, in fact remove this parameter completely from the program, but we will leave it in for now, so we do not have to modify the LINKAGE SECTIONS of the three methods in the FILEIO3 program. We no longer need this parameter because we are calling specific methods to OPEN, READ, and CLOSE the file.
2. We have removed the commented-out CALL statements for clarity.
3. We have changed the Invoke statement to the "CALL_FILEIO2" method to instead Invoke the 3 specific methods in FILEIO3.

We now have a much more object-oriented FILEIO3 program and we can see that even though we have made FILEIO3 a bit larger in number of lines of source code than the previous FILEIO2 program, the TESTFILEIO3 program that calls it is much simpler. This is on of the benefits of using Object-Oriented COBOL.

In the final example we will clean up the application even more and create a final polished and well-structured object-oriented FILEIO4 program.

2.3 Converting the FILEIO3 OO COBOL Program into a Well-structured Object-Oriented COBOL Program

While the FILEIO3 OO COBOL program works perfectly fine, there is some cleanup we can perform to make it even more efficient and easier to maintain.

Specifically, we can remove the exit labels and exit statements from the end of each of the three methods (these were carryovers from when these methods used to be paragraphs). Additionally we can customize the three methods to use only the parameters they need.

Figure 2.8 shows the final FILEIO4 program which is the FILEIO3 program having been cleaned up.

```
Identification Division.
Class-ID. FILEIO4 Inherits FJBASE.
Environment Division.
Configuration Section.
Repository.
    Class FJBASE.

Object.
Environment Division.
Input-Output Section.
File-Control.
    Select Infile Assign To "test.txt"
                  Organization is Line Sequential
                  File Status is Infile-Status.
```

```
Data Division.
File Section.
FD Infile.
01 Infile-Record Pic X(132).
Working-Storage Section.
01 Infile-Status Pic 99.
01 File-Opened-Flag Pic 9 Value 0.
Procedure Division.

Method-ID. OPEN-FILE.
Data Division.
Linkage Section.
01 Return-Status Pic 99.
Procedure Division Using Return-Status.
    If File-Opened-Flag Not = 0
        *> File is already open, so return error
        Move 99 To Return-Status
    Else
        Open Input Infile
        Move Infile-Status To Return-Status
        If Infile-Status = 0
            Move 1 To File-Opened-Flag
        End-If
    End-If.
End Method OPEN-FILE.

Method-ID. CLOSE-FILE.
Data Division.
Linkage Section.
01 Return-Status Pic 99.
Procedure Division Using Return-Status.
    If File-Opened-Flag = 0
        *> File is not open, so return error
        Move 88 To Return-Status
    Else
        Close Infile
        Move Infile-Status To Return-Status
        Move 0 To File-Opened-Flag
    End-If.
End Method CLOSE-FILE.

Method-ID. READ-FILE.
Data Division.
Linkage Section.
01 Return-Status Pic 99.
01 Return-Record Pic X(132).
Procedure Division Using Return-Status
Return-Record.
    If File-Opened-Flag = 0
        *> File is not open, so return error
        Move 88 To Return-Status
    Else
        Read Infile
        Move Infile-Status to Return-Status
        Move Infile-Record to Return-Record
    End-If.
End Method READ-FILE.

End Object.
End Class FILEIO4.
```

Figure 2.8 The FILEIO4 program

In comparing the FILEIO4 program shown in Figure 2.8 to the FILEIO3 program shown in Figure 2.6, we note the following modifications:

1. We have removed the exit labels and exit statements contained at the bottom of each of the three methods.
2. We have altered the LINKAGE SECTIONS and associated PROCEDURE DIVISION USING … statements to reflect only the parameters required by the method. The OPEN-FILE and CLOSE-FILE methods do not need the RETURN-RECORD parameter because they don't return a record. We have also removed the FILE-OPERATION parameter from all three methods because it is no longer required.

We must now modify the TESTFILEIO3 program to reflect the changes in the parameter structures of the three methods. This makes this program even shorter and less complicated. The resulting TESTFILEIO4 program is shown in Figure 2.9.

```
Identification Division.
Program-ID. TESTFILEIO4.
Environment Division.
Configuration Section.
Repository.
    Class FILEIO4.
Data Division.
Working-Storage Section.
01 FILEIO-PTR Object Reference.
01 Return-Status Pic 99 Value 0.
01 Return-Record Pic X(132).
Procedure Division.
    Invoke FILEIO4 "NEW" Returning FILEIO-PTR.
    Invoke FILEIO-PTR "OPEN-FILE" Using Return-Status.
    If Return-Status = 0
        Perform Until Return-Status Not = 0
            Invoke FILEIO-PTR "READ-FILE" Using Return-Status
                                              Return-Record
            If Return-Status = 0
                Display Return-Record
            End-If
        End-Perform
        Invoke FILEIO-PTR "CLOSE-FILE" Using Return-Status
    End-If.
    Set FILEIO-PTR To Null.
    Exit Program.
```

Figure 2.9 The TESTFILEIO4 program

We now have a well-structured FILEIO4 OO COBOL program. You should now have some understanding of the issues involved and the process required for modifying an existing traditional procedural COBOL program into an OO COBOL program

In the next chapter, we will take the FILEIO4 program and create a Microsoft Windows COM component out of it. We will then illustrate how this component can be used in a Microsoft Visual Basic® (VB) application. Finally, we will create a simple Internet-based Active Server Page (ASP) application that uses our FILEIO4 COM component.

Chapter 3 Creating and Using COBOL COM Components

In this chapter, we will look at the steps involved in taking the FILEIO4 OO COBOL program created in the previous chapter and turning it into a COM component.

We are going use the FILEIO4 program to create a COM Server module. A COM server module is a program that expects to be called from another COM Client program.

We will modify the TESTFILEIO4 program into a COM client to illustrate how COBOL programs can instantiate COM Server modules.

We first need to create a project named FILEIO5.PRJ. Then we will build a TESTFILEIO5.EXE program and a FILEIO5.DLL. As a beginning point, we will create TESTFILEIO5.COB and FILEIO5.COB source programs by copying the current TESTFILEIO4.COB and FILEIO4.COB programs.

3.1 Creating a COM Server COBOL Program

Figure 3.1 shows the initial FILEIO5.PRJ project.

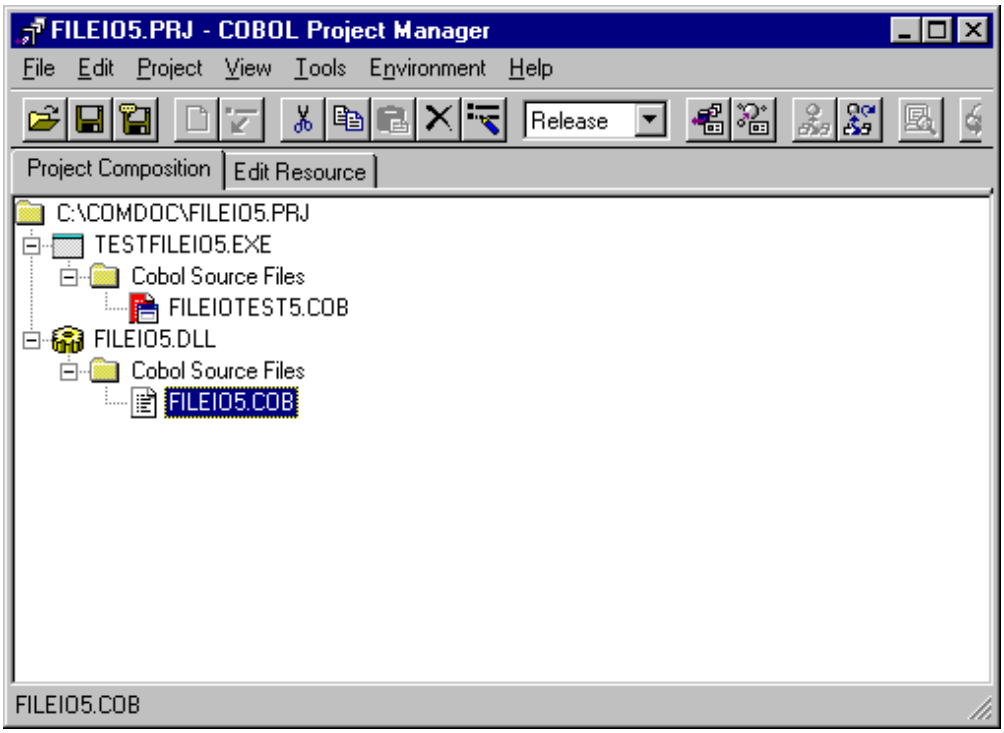

Figure 3.1 The initial FILEIO5.PRJ project

We will need to add a Target Repository Files folder and add the FILEIO5.REP file to this folder under FILEIO5.COB in the project hierarchy. This file is generated by the NetCOBOL compiler and is required when FILEIO5 is built.

We create the Target Repository Files folder by right clicking on the FILEIO5.COB file, selecting Create Folder from the context menu that appears, and then selecting the Target Repository Files option.

To add the FILEIO5.REP file under the newly created Target Repository Files folder, right click on the folder and select the Add File option. The FILEIO.REP file should be displayed in the file list that appears and you can select it. Note that this file will not exist unless you have already successfully built the FILEIO5.DLL program. In this case, you may select either the “All” option to add all repository files, or the “Target” option to add only the targeted repository file once it is created.

Figure 3.2 shows the project hierarchy with the FILEIO5.REP file properly added.

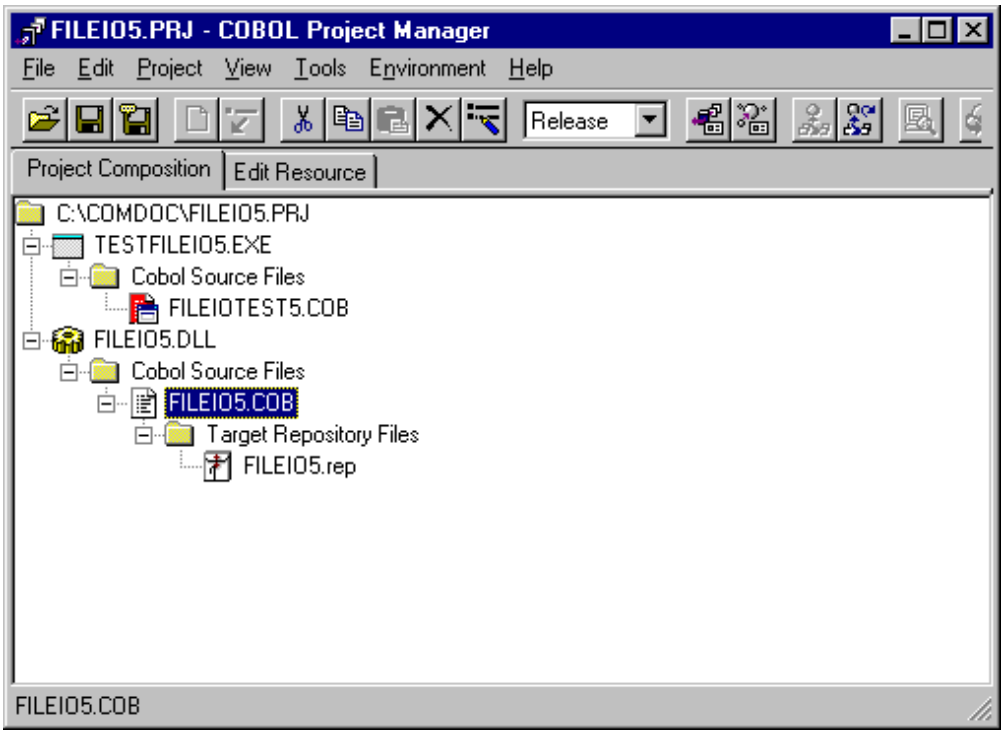

Figure 3.2 The project hierarchy with the FILEIO5.REP file added.

Once the FILEIO5.DLL has been successfully built and the FILEIO5.REP file has been added to the project hierarchy, we are ready to specify creation of a COM Server module for FILEIO5.DLL.

To accomplish this, we highlight the FILEIO5.DLL file in the project hierarchy and select the Project pulldown menu, move to the Option item and then to COM Server on the context menu that appears. We then select the Edit option as shown in Figure 3.3.

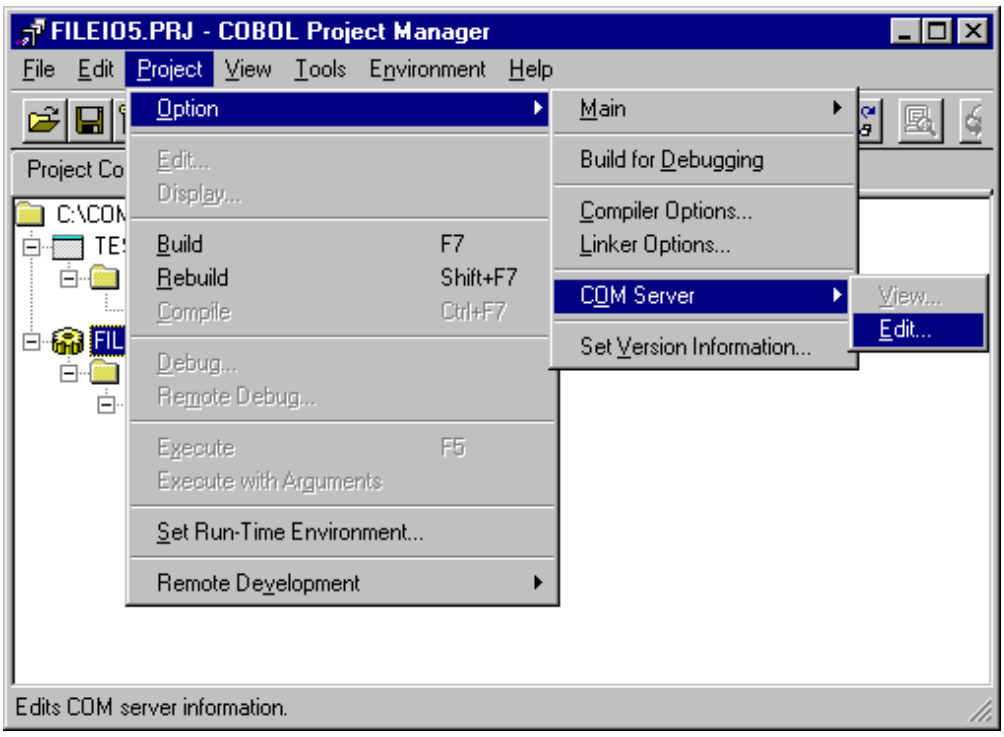

Figure 3.3 Selecting the Edit option of COM Server Specification

This brings up the Com Server Edit dialog box as shown in Figure 3.4.

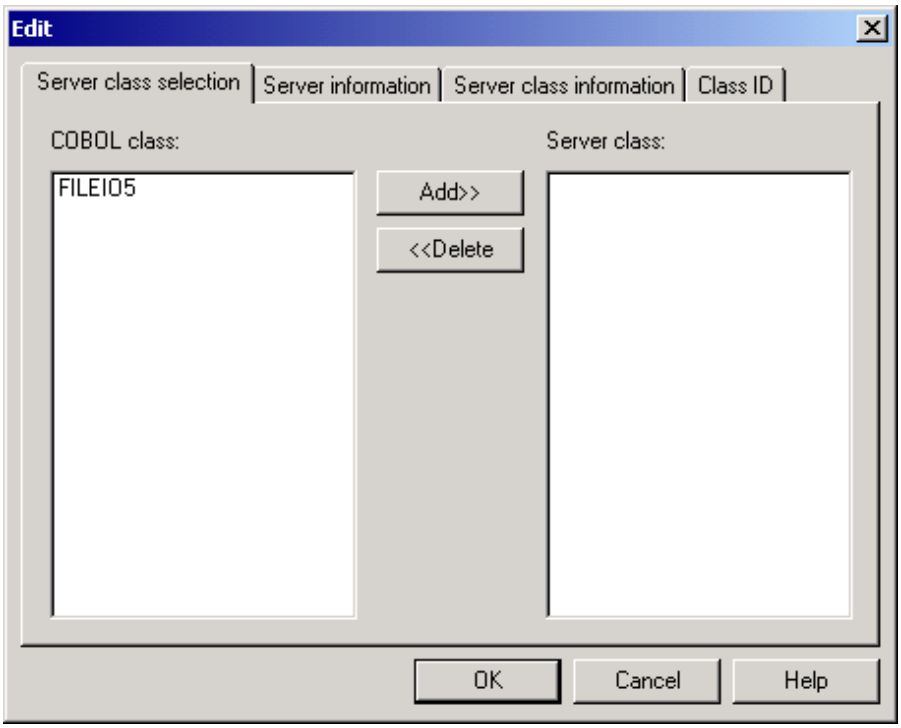

Figure 3.4 The COM Server specification Edit dialog box

Highlight the FILEIO5 class in the COBOL class: Option box and click on the ADD>> button to add it to the Com Server Class. Then click on the OK button to apply this selection and close the dialog box.

You will now notice that a new icon has appeared to the left of the FILEIO5.DLL file in the project hierarchy window as shown in Figure 3.5.

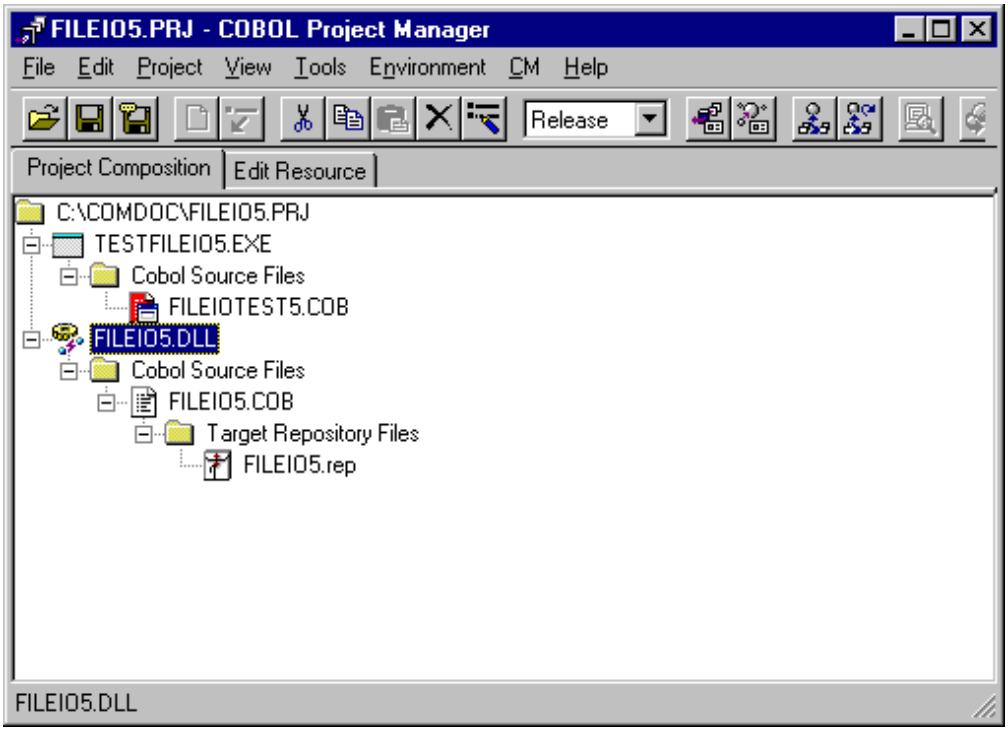

Figure 3.5 The Com Server Icon displayed next to FILEIO5.DLL

3.2 Errors Caused by using Hyphens in the COM Interface

After adding the FILEIO5.Rep file to the project hierarchy, you should attempt to rebuild the FILEIO5.DLL file. When doing so, the following error will be encountered:

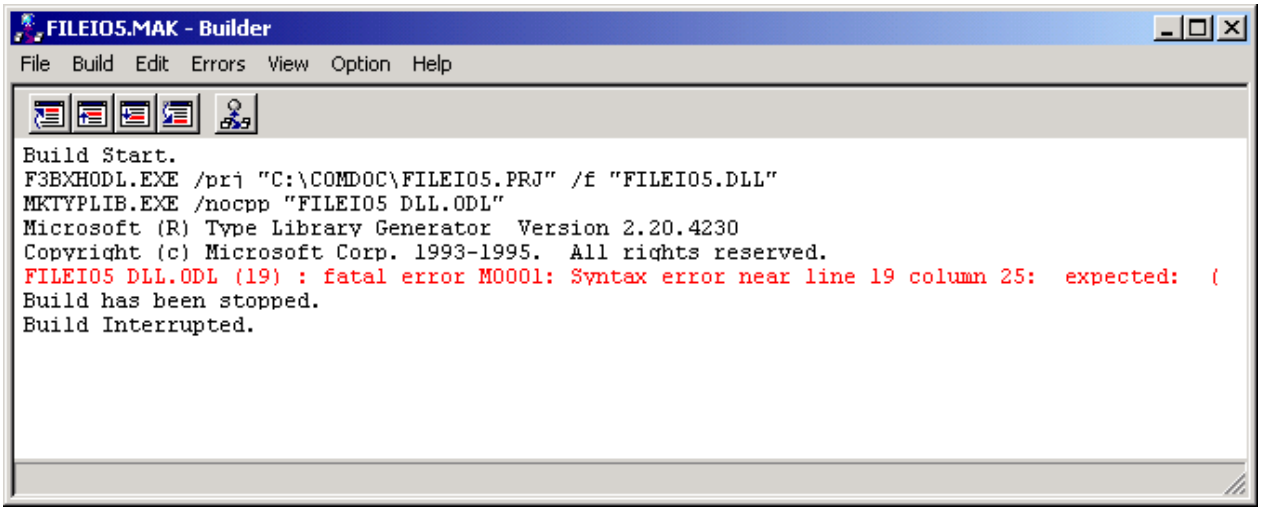

Figure 3.6 An error displayed when trying to build the FILEIO5.DLL program

The message displayed in the Builder window shown in Figure 3.6 is somewhat mysterious and warrants some explanation. When you create a COM Server FILEIO5.DLL, one of the files created with be named FILEIO5_DLL.ODL.

This .ODL contains a special interface definition language used by COM. The file created in this instance is shown in Figure 3.7.

```
// FILEIO5_DLL.odl
[
    uuid(9427D108-7ACB-4937-92CE-E143D9B4AC42),
    helpstring("FILEIO5.DLL"),
    version(1.0)
]

library FILEIO5
{
    importlib("stdole2.tlb");
    dispinterface _FILEIO5;
    [uuid(9D47C2A1-DF64-4710-BE9D-2BFA6FB930D9)]
    dispinterface _FILEIO5
    {
        properties:
        methods:
        [id(1)] void CLOSE-FILE([in,out] decimal* RETURN-STATUS);
        [id(2)] void OPEN-FILE([in,out] decimal* RETURN-STATUS);
        [id(3)] void READ-FILE([in,out] decimal* RETURN-STATUS,[in,out] bstr* RETURN-RECORD);
        [id(4)] void INIT();
        [id(5)] void _FINALIZE();
    };

    [
        uuid(1A97EC50-A138-461C-BA4F-88BD0A65136C),
        version(1.0)
    ]

    coclass FILEIO5
    {
        [default] dispinterface _FILEIO5;
```

```
    };
};
```

Figure 3.7 The FILEIO5.DLL.ODL file

The error message shown in Figure 3.6 notes an error on line 19 at position 25. Line 19 in this .ODL file contains:

```
[id(1)] void CLOSE-FILE([in,out] decimal* RETURN-STATUS);
```

If you count over 25 columns, you will find the hyphen character in the CLOSE-FILE method name.

This error is complaining about the use of a hyphen (“-“) in the COM interface. Hyphens are not allowed in the COBOL TO COM interface, so we must find every place a hyphen appears in either a method name or a LINKAGE SECTION data item name being passed to or from a method and change it. An underscore (“_”) is a good substitute.

This means we need to edit the FILEIO5.COB program and change the Method names and their LINKAGE SECTION parameter names from containing any hyphens to underscores.

The results of our changes are shown in Figure 3.8.

```
Identification Division.
Class-ID. FILEIO5 Inherits FJBASE.
Environment Division.
Configuration Section.
Repository.
    Class FJBASE.

Object.
Environment Division.
Input-Output Section.
File-Control.
    Select Infile Assign To "test.txt"
                  Organization is Line Sequential
                  File Status is Infile-Status.
Data Division.
File Section.
FD Infile.
01 Infile-Record    Pic X(132).
Working-Storage Section.
01 Infile-Status    Pic 99.
01 File-Opened-Flag Pic 9 Value 0.
Procedure Division.

Method-ID. OPEN_FILE.
Data Division.
Linkage Section.
01 Return_Status    Pic 99.
Procedure Division Using Return_Status.
    If File-Opened-Flag Not = 0
        *> File is already open, so return error
        Move 99 To Return_Status
    Else
        Open Input Infile
        Move Infile-Status To Return_Status
        If Infile-Status = 0
            Move 1 To File-Opened-Flag
        End-If
    End-If.
End Method OPEN_FILE.

Method-ID. CLOSE_FILE.
Data Division.
Linkage Section.
```

```
01 Return_Status    Pic 99.
Procedure Division Using Return_Status.
    If File-Opened-Flag = 0
        *> File is not open, so return error
        Move 88 To Return_Status
    Else
        Close Infile
        Move Infile-Status To Return_Status
        Move 0 To File-Opened-Flag
    End-If.
End Method CLOSE_FILE.

Method-ID. READ_FILE.
Data Division.
Linkage Section.
01 Return_Status    Pic 99.
01 Return_Record    Pic X(132).
Procedure Division Using Return_Status
                         Return_Record.
    If File-Opened-Flag = 0
        *> File is not open, so return error
        Move 88 To Return_Status
    Else
        Read Infile
        Move Infile-Status to Return_Status
        Move Infile-Record to Return_Record
    End-If.
End Method READ_FILE.

End Object.
End Class FILEIO5.
```

Figure 3.8 The corrected FILEIO5 program having changed hyphens to underscores

Note that in making these changes you do not have to change all data items names to get rid of hyphens - only those that are used in the interface as LINKAGE SECTION items. You must also change method names and any public property names (we do not have any public property names in this example, however).

Now we can attempt to build this application again and we do not receive any errors.

3.3 Registering Your COM Module

You are now ready to register the .DLL using the Windows Regsvr32.exe program. You can do this by going to a DOS command line and typing:

```
C:\COMDOC> Regsvr32 FILEIO5.DLL
```

Where: C:\COMDOC is the directory containing your FILEIO5.DLL file.

If you are going to work with COBOL COM modules often, you may wish to add an entry to call Regsvr32.Exe from the COBOL Project Manager Tools pulldown menu.

You can do this by selecting the Customize Menu option under the Tools pulldown menu, and then clicking on the Add button in the dialog box that appears. Give it the name Regsvr32 in the Name on Menu field. You then choose the Regsvr32.Exe file by navigating to the C:\Windows\System32 subdirectory (or C:\WINNT\SYSTEM32 under Windows NT and Windows 2000) and selecting it in the Execution File field. Make sure you also select the "Selected Files" option in the Arguments field. This will allow you to select a .DLL file in the project hierarchy and then select the Regsvr32 option you added to the Tools pulldown menu and it will feed the name of the selected .DLL file into Regsvr32.

One you execute Regsvr32.exe on FILEIO5.DLL, you should receive the following dialog box indicating a successful registration.

Figure 3.9 The Regsvr32.EXE successful registration dialog box

You are now ready to use FILEIO5.DLL as a COM component. We will next modify the TESTFILEIO5 program to be a COM Client that will invoke the FILEIO5.DLL COM component.

3.4 Creating a COM Client Program

We will now modify the TESTFILEO5 program to be a COM client that will invoke the FILEIO5 COM Server program.

COBOL COM Client programs are not limited to invoking COBOL COM Server programs. Instead they can invoke COM Server programs written in a wide variety of languages such as Visual Basic and Visual C++. The technique shown below will work for COM Server modules written in other languages as well.

The TESTFILEIO5 program has been modified as shown in Figure 3.10 to turn it into a COM Client program.

```
Identification Division.
Program-ID. TESTFILEIO5.
Environment Division.
Configuration Section.
Repository.
    Class FILEIO5 AS "*COM".
Data Division.
Working-Storage Section.
01 FILEIO-PTR Object Reference FILEIO5.
01 Return_Status Pic 99 Value 0.
01 Return_Record Pic X(132).
01 Prog-ID Pic X(15) Value "FILEIO5.FILEIO5".
Procedure Division.
    Invoke FILEIO5 "CREATE-OBJECT" Using Prog-ID
                                   Returning FILEIO-PTR.
    Invoke FILEIO-PTR "OPEN_FILE" Using Return_Status.
    If Return_Status = 0
        Perform Until Return_Status Not = 0
            Invoke FILEIO-PTR "READ_FILE" Using Return_Status
                                               Return_Record
            If Return_Status = 0
                Display Return_Record
            End-If
        End-Perform
        Invoke FILEIO-PTR "CLOSE_FILE" Using Return_Status
    End-If.
    Set FILEIO-PTR To Null.
    Exit Program.
```

Figure 3.10 The TESTFILEIO5 program modified to be a COM Client program

If you compare the COM Client program TESTFILEIO5 shown in Figure 3.10 to its predecessor (TESTFILEIO4) shown in Figure 2.9, you will notice the following modifications were made:

1. The Class reference statement in the Repository was changed to:

```
Class FILEIO5 AS "*COM".
```

The *COM class is a special class provided by NetCOBOL that allows COBOL programs to invoke COM classes. This statement identifies FILEIO5 as a COM class.

2. The FILEIO-PTR data item's definition has been changed from an Object Reference to:

```
01 FILEIO-PTR Object Reference FILEIO5.
```

This identifies this Object Reference as an object reference that will point at a FILEIO5 class instance at runtime.

3. We've added a new data item to the WORKING-STORAGE SECTION:

```
01 Prog-ID Pic X(15) Value "FILEIO5.FILEIO5".
```

When Regsvr32.Exe registered the FILEIO5 COM Server module, it created an entry in the Windows registry for it entitled "FILEIO5.FILEIO5". This name was derived from the actual .DLL name (FILEIO5) and the name of the Class we specified for the COM Server (FILEIO5). This name will be used at runtime to create an instance of it.

4. The first Invoke statement has been changed to:

```
Invoke FILEIO5 "CREATE-OBJECT" Using Prog-ID Returning FILEIO-PTR.
```

The "CREATE-OBJECT" method was inherited from the FJBASE class. It creates an object at runtime using Late Binding. We will discuss Late Binding vs. Early Binding a bit later in this chapter. This call uses the Prog-ID data item and returns a pointer (object reference) to a FILEIO5 object.

You will notice that all of the remaining procedural code remains the same, and if you build and execute the TESTFILEIO5 program, you will receive the same results as in all of the previous projects we've used in this manual.

3.5 Early Binding vs. Late Binding

As noted previously, the TESTFILEIO5 program has been modified to use Late Binding. We could have made the TESTFILEIO5 program use Early Binding instead making two changes to the program:

1. The Class reference statement in the Repository would be changed to:

```
Class FILEIO5 As "*COM:FILEIO5:FILEIO5".
```

2. The first invoke statement that currently calls the "CREATE-OBJECT" method would be changed to:

```
Invoke FILEIO5 "NEW" Returning FILEIO-PTR.
```

Early Binding requires class definitions specified in the REPOSITIORY section to be written in the format:

```
"*COM:com-server-name:class-name"
```

The com-server-name can be any name, but the class-name must be found in the associated type library, which is required for early binding. The com-server-name can be freely assigned by the user, but must be associated with the location of the type library and stored in the option file.

But what exactly is Early Binding vs. Late Binding? Binding defines the mode in which the client checks a method or interface of a COM server to determine interface specifics such as parameter types. This check must be done to ensure that the client is calling a valid method of the COM Server and that it is passing the correct number of properly defined parameters.

This check can take place at build time (similar in nature to a Linkage Editor resolving external references), or it can be done at dynamically at runtime. Performing this check at build time is known as Early Binding. Performing this check dynamically at runtime is known as Late Binding.

Late Binding has a slight performance penalty because additional work is required at runtime, but it has the benefit of allowing you to change COM Server .DLLs and rebuild them without being forced to rebuild the COM clients that access them.

Early Binding has the benefit of a slight performance gain because this work is done at build time and not required at runtime. It also has the benefit of finding interface errors at build time as opposed having to wait to find these types of errors at runtime.

Early binding does require the existence of a type library for any COM Server you wish to call, whereas Late Binding does not have this requirement.

Unless you are in need of every bit of performance you can obtain, Late Binding is typically a better option - especially when doing development as modules change often.

You should now have a basic understanding of how to create both COM Client and COM Server applications using NetCOBOL.

In the next chapters we will look at invoking the FILEIO5 COM server module from both Visual Basic and from an Internet application using an ASP page.

Chapter 4 Using COBOL COM Servers with Visual Basic

Now that we have the FILEIO5 COM Server module created and registered we would like to use it from Visual Basic.

Unfortunately, once we step outside the COBOL world, we will find that under Visual Basic, we will receive errors if we try to use the FILEIO5 COM Server Program. We need to make some modifications and these are shown in the updated FILEIO5 program in Figure 4.1.

```
Identification Division.
Class-ID. FILEIO5 Inherits FJBASE.
Environment Division.
Configuration Section.
Repository.
    Class FJBASE.

Object.
Environment Division.
Input-Output Section.
File-Control.
    Select Infile Assign To "c:\comdoc\test.txt"
                  Organization is Line Sequential
                  File Status is Infile-Status.
Data Division.
File Section.
FD Infile.
01 Infile-Record Pic X(132).
Working-Storage Section.
01 Infile-Status Pic 99.
01 File-Opened-Flag Pic 9 Value 0.
Procedure Division.

Method-ID. OPEN_FILE.
Data Division.
Linkage Section.
01 Return_Status Pic S9(9) COMP-5.
Procedure Division Returning Return_Status.
    If File-Opened-Flag Not = 0
        *> File is already open, so return error
        Move 99 To Return_Status
    Else
        Open Input Infile
        Move Infile-Status To Return_Status
        If Infile-Status = 0
            Move 1 To File-Opened-Flag
        End-If
    End-If.
End Method OPEN_FILE.

Method-ID. CLOSE_FILE.
Data Division.
Linkage Section.
01 Return_Status Pic S9(9) COMP-5.
Procedure Division Returning Return_Status.
    If File-Opened-Flag = 0
        *> File is not open, so return error
        Move 88 To Return_Status
    Else
        Close Infile
        Move Infile-Status To Return_Status
        Move 0 To File-Opened-Flag
    End-If.
End Method CLOSE_FILE.
```

```
Method-ID. READ_FILE.
Data Division.
Linkage Section.
01 Return_Record Pic X(132).
01 Return_Status Pic S9(9) COMP-5.
Procedure Division Using Return_Record
                  Returning Return_Status.
    If File-Opened-Flag = 0
        *> File is not open, so return error
        Move 88 To Return_Status
    Else
        Read Infile
        Move Infile-Status to Return_Status
        Move Infile-Record to Return_Record
    End-If.
End Method READ_FILE.

End Object.
End Class FILEIO5.
```

Figure 4.1 Updated FILEIO5 program for use with Visual Basic

If you compare the FILEIO5 program in Figure 4.1 with the previous version shown in Figure 3.8, you will note the following modifications:

1. The name of the text file being accessed in the SELECT⋯ ASSIGN to statement has had a path added to it as shown:

   ```
   Select Infile Assign To "c:\comdoc\test.txt"
   ```

   The reason for this is that Visual Basic changes directories at runtime time and you cannot be sure that you will remain positioned in the same directory as you started execution in. This causes the file note to be found when attempting to open it with a path name on it.

2. The RETURN_STATUS data item has had its definition changed from Pic 99 to Pic S9(9) COMP-5. This is an important modification. While we were executing completely in COBOL, we had the luxury of using any COBOL data type we desired. COM, however, supports a fixed set of specialized data types and Visual Basic does not support all of the various COBOL data types as a result. This means we must choose data types as parameters in method calls that are supported by the language that the COM client is written in. In this case, the Pic S9(9) COMP-5 is a LONG in Visual Basic, so we'll use this definition for the RETURN_STATUS data item in all three of our methods.

3. PROCEDURE DIVISION USING ⋯ statements in the OPEN_FILE and CLOSE_FILE methods have been changed to PROCEDURE DIVISION Returning ⋯ This signals the COM interface that we expect to return the data item named RETURN_STATUS to the COM client. It also allows us to call these methods from Visual Basic in a more logical fashion as well see a bit later in this chapter.

4. The LINKAGE SECTION parameters in the Read_File method have been reordered so that we can change the PROCEDURE DIVISION USING⋯ statement to:

   ```
   Procedure Division Using Return_Record
                     Returning Return_Status.
   ```

   This allows us to code the call to this method from Visual Basic in a more logical fashion, as we'll see a bit later in this chapter.

Once we've applied these previously noted changes to our FILEIO5 module, rebuilt it and re-registered it using Regsvr32.exe, we are ready to create a Visual Basic project to access it.

To illustrate this, we create a standard .EXE project in Visual Basic and create a simple form with a command button (push button) control and a list box control. We will then implement behavior such that when the user clicks on the push button, the Visual Basic program creates an instance of the FILEIO5 class and executes its OPEN_FILE, READ_FILE, and CLOSE-FILE methods. Each record returned from the READ_FILE method will be added to the list box control for display purposes.

The Visual Basic form in this application is shown in Figure 4.2.

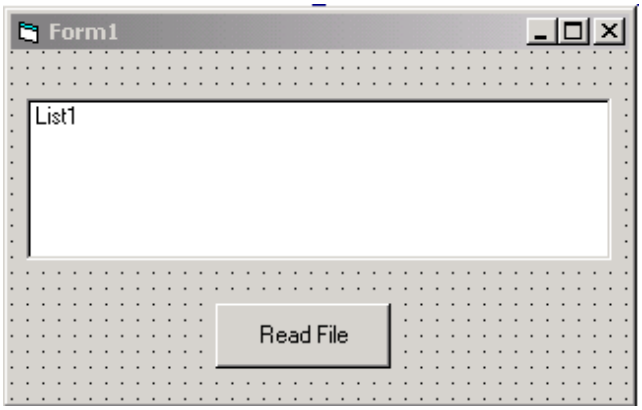

Figure 4.2 The Visual Basic form

Before we write any code referencing the FILEIO5 COM Server module, we must first add a reference to it in our Visual Basic application. This is done by selecting the References option in the Project pulldown menu in Visual Basic as shown in Figure 4.3.

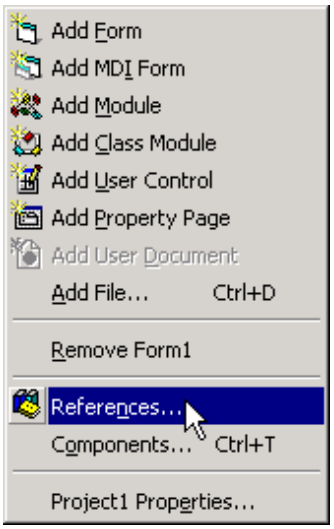

Figure 4.3 The References option in the Project pulldown menu

This brings up the References dialog box listing all of the references available on the current system. You must scroll down and find the FILEIO5.DLL reference and check it as shown in Figure 4.4.

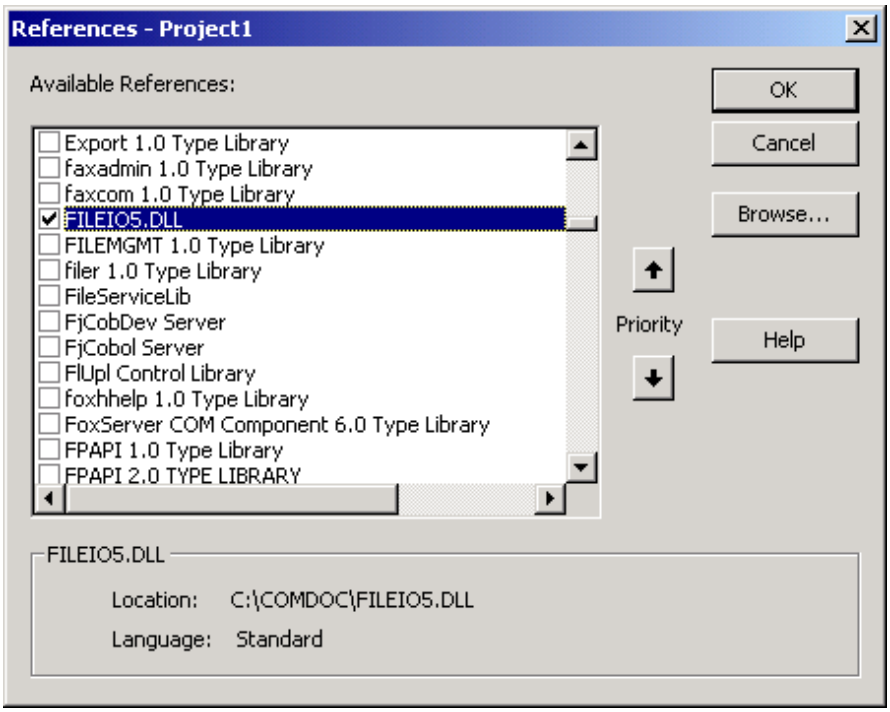

Figure 4.4 The References dialog box

After checking the FILEIO5.DLL check box, click on the OK button to apply this and close the Reference dialog box. We are now ready to write the actual Visual Basic code to implement the previously described behavior.

The source code of our Visual Basic Project is shown in Figure 4.5. We need only implement code for the Click event for the command button control named Command1.

```
Private Sub Command1_Click()
Dim Return_Status As Long
Dim Return_Record As String
Dim Return_Code As Long
Dim FILEIO_PTR As FILEIO5.FILEIO5
    Set FILEIO_PTR = New FILEIO5.FILEIO5
    Return_Status = FILEIO_PTR.OPEN_FILE()
    If Return_Status = 0 Then
        Do While Return_Status = 0
            Return_Status = FILEIO_PTR.READ_FILE(Return_Record)
            If Return_Status = 0 Then
                List1.AddItem (Return_Record)
            End If
        Loop
        Return_Status = FILEIO_PTR.CLOSE_FILE()
    End If
    Set FILEIO_PTR = Nothing
End Sub
```

Figure 4.5 Visual Basic project source code

If we execute the previously noted Visual Basic application, the results will be as shown in Figure 4.6.

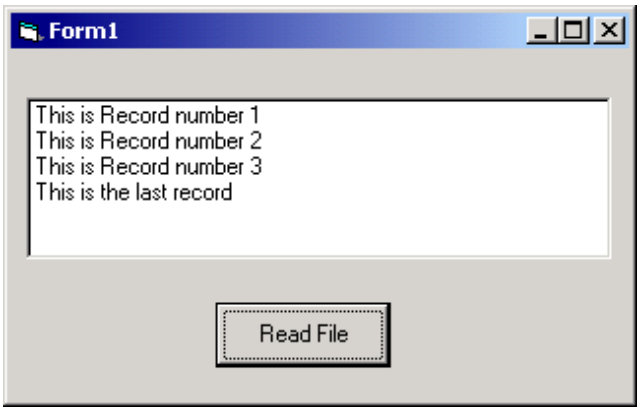

Figure 4.6 The results of executing the Visual Basic application

You should now understand how to create a Visual Basic COM Client program that accesses a COBOL COM Server program. In the final chapter, we will create an Internet application using an ASP page that calls the FILEIO5 COM Server program.

Chapter 5 Using COBOL COM Servers with ASP Pages

In this final chapter, we'll look at how to create an Active Server Page (ASP) that will invoke our FILEIO5 COM Sever module from a Web Browser.

Because Visual Basic Script is the default scripting engine for ASP pages, our previous Visual Basic experience will prove invaluable in creating an ASP page.

Creating an ASP page to instantiate the FILEIO5 class and invoke its methods is a very straightforward process.

The resulting ASP page will be named "fujcom.asp" and is shown in Figure 5.1.

```
<% Language=VBScript%>
<%
Set FILEIO_PTR = Server.CreateObject("FILEIO5.FILEIO5")
    Return_Status = FILEIO_PTR.OPEN_FILE()
    If Return_Status = 0 Then
        String1 = "<P>"
        Do While Return_Status = 0
        Return_Status = FILEIO_PTR.READ_FILE(Return_Record)
        If Return_Status = 0 Then
            String1 = String1 & Return_Record & "</P><P>"
        End If
        Loop
        Return_Status = FILEIO_PTR.CLOSE_FILE()
    End If
    Set FILEIO_PTR = Nothing
%>

<HTML>
<HEAD>
<TITLE>NetCOBOL COM Server Example</TITLE>
</HEAD>

<BODY>
<H3><Center>Fujtisu COBOL COM Server Returned:</Center></H3>
<Center> <% Response.Write String1 %> </Center>
</BODY>
</HTML>
```

Figure 5.1 The fujcom.asp page's source code

If you review the ASP page's source code shown in Figure 5.1, you'll note that it is similar to the source code created in the Visual Basic project in the previous chapter with the following exceptions:

1. VB Script is the language in use in the ASP page. VB Script automatically defines and allocates data types, so DIM statements are not required.
2. We call the FILEIO5 COM Server's methods in the same manner we called them from our previous Visual Basic project.
3. The loop logic has been changed to create a single string named "String1" and to append each read record onto the end of it with appropriate HTML syntax to create a new paragraph for each record read.
4. Once all records are read and concatenated onto the String1 data item using the READ_FILE method, we close the file by calling the CLOSE_FILE method. We then release the FILEIO5 object by setting its pointer to nothing, just as we did in the previous Visual Basic application.
5. We use the VB script method Response.Write to write our string out in the HTML code that comes after we exit the VB Script routine we coded.

To execute this ASP page, copy it to the root directory of your Internet Server. This is typically the directory named:

```
C:\inetpub\wwwroot\
```

Bring up your web browser and enter:

http://www.my.com/fujcom.asp

where: www.my.com is the name of your web domain. You can optionally enter:

http://111.111.111.111/fujcom.asp

where: 111.111.111.111 is the IP address of your web server.

When you execute the fujcom.asp page, you will see the following display in your Web Browser.

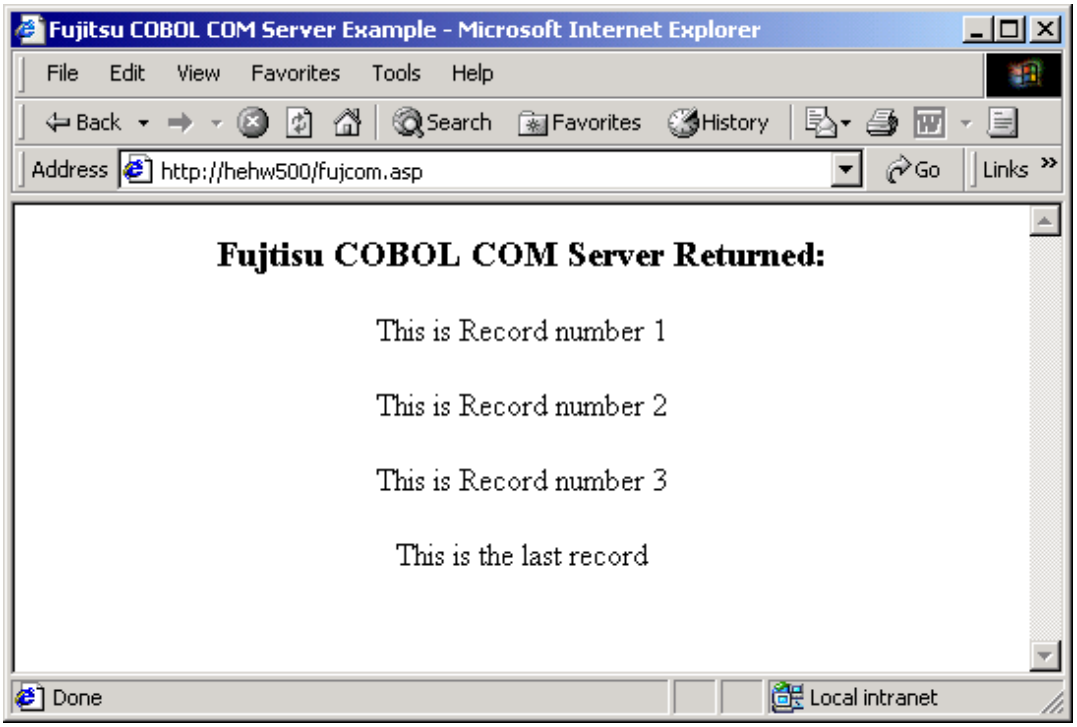

Figure 5.2 The fujcom.asp page in a Web Browser

You should now have an understanding of how to instantiate a COBOL COM Server module from an ASP page and access its methods.